Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 2

Week: 02

Due: Oct 27, 2019

**Lab 2 - Solutions**

Note: Java source code is on <https://repl.it/@QuanDoan/AlgoLabs>, Lab2.java

**Problem 1**:

Number of primitive operation is considered as n.

|  |  |  |
| --- | --- | --- |
| A | sum 🡨 0  for i 🡨 0 to n – 1 do  sum 🡨 sum + 1 | 🡺 4 |
| B | sum 🡨 0  for i 🡨 0 to n – 1 do  for j 🡨 0 to n – 1 do  sum 🡨 sum + 1 | 🡺 |

**Problem 2**:

The time complexity of the below procedure is .

|  |  |
| --- | --- |
| int[] arrays(int n) {  int[] arr = new int[n];  for(int i = 0; i < n; ++i) {  arr[i] = 1;  }  for(int i = 0; i < n; ++i) {  for(int j = i; j < n; ++j) {  arr[i] += arr[j] + i + j;  }  }  return arr;  } | (assignments for array of n numbers)  🡺 |

**Problem 3**:

1. Algorithm

The idea is to use 3 separate loops, which is O(n) complexity, rather than using nested loop.

* Allocate an array arr3, which has size of arr1 and arr2
* Loop through arr1 and arr2 in the same time
  + Compare the current element from arr1 and arr2, then copy the smaller to next position in arr3, then move ahead in arr3 and the array which has the picked element.

For example, element 3rd of arr1 is smaller than element 5th of arr2, then that element 3rd will be selected and put in the next position in arr3, then the index increased by 1.

* The remaining elements in arr1 or arr2 will be copied to arr3.

1. Pseudo-code

|  |  |
| --- | --- |
| Input: array 1 and array 2 sorted  Output: array 3  array3 🡨 new arr, length = arr1 length + arr2 length  i 🡨 0  j 🡨 0  k 🡨 0  while i < arr1 length and j < arr2 length do  If arr1[i] < arr2[j] then  arr3[k++] 🡨 arr1[i++]  else  arr3[k++] 🡨 arr2[j++]  while i < arr1 length do  arr3[k++] 🡨 arr1[i++]  while j < arr2 length do  arr3[k++] 🡨 arr2[j++]  return arr3 | 1  1  1  1  n1 + n2  n1  n1  n2  2.n1  n1  2.n2  n2  🡺 f(n1)+g(n2) = O(n1+n2) |

1. Java code

The implementation does not return result as an array, instead it prints out its result.

// Question 3

public static void mergeTwoSorted(int[] arr1, int[] arr2) {

int[] arr3 = new int[arr1.length + arr2.length];

int i = 0;

int j = 0;

int k = 0;

while (i < arr1.length && j < arr2.length) {

if (arr1[i] < arr2[j]) {

// first: the value will be from k and i,

// then k and i will be increased by 1

// right after the execution of the line.

arr3[k++] = arr1[i++];

} else {

arr3[k++] = arr2[j++];

}

}

while (i < arr1.length) {

arr3[k++] = arr1[i++];

}

while (j < arr2.length) {

arr3[k++] = arr2[j++];

}

System.out.println("Arr 1: ");

printArray(arr1);

System.out.println("Arr 2: ");

printArray(arr2);

System.out.println("Merged array: ");

printArray(arr3);

}

**Problem 4**:

1. is : true

is same as , or , which is

If n = 0 or n = 1, f(n) == 0

If n >= 2, f(n) <

1. is not : true

False.

If n is increasing more than 2 (for example), then is similar as , which is .

1. is : False

If n = 1, log 1 = 0

If n = 0, log 0 is undefined, cannot compare. Hence, for all n > 1, n < n + < 2n, which is .

Meaning is not

1. is not : False

If n = 1 > n = 0, which is not the little-o, if for all arbitrarily small real c > 0, for all but perhaps finitely many n,

Equivalently, f is if the function tends to 0 as n tends to infinity. That is, f is small compared to g. If f is then f is also .

Actually, is .

**Problem 5**:

To prove , we need to find constants k and such that for all n ≥ , .

Since:

and

then by definition:

So, , and

Then , for all .

Hence .

**Problem 6**:

Java code:

// Question 6

// PowerSet: Time Complexity: O( 2^n )

// This iterative is the implementation of the given algo in Lab 2.

// Each subset is in the order that the element appears in the input list.

// With this implementation the input is kept separately.

public static <T> List<List<T>> powerset(Collection<T> list) {

List<List<T>> ps = new ArrayList<List<T>>();

ps.add(new ArrayList<T>()); // add empty set

// for every item in the original list

for (T item : list) {

// A temporary holder for the generated powerset

List<List<T>> tempPs = new ArrayList<List<T>>();

for (List<T> subset : ps) {

// copy all items from the current powerset's subsets

tempPs.add(subset);

// add the subsets appended with the current item

List<T> newSubset = new ArrayList<T>(subset);

newSubset.add(item);

tempPs.add(newSubset);

}

ps = tempPs;

}

return ps;

}

**Problem 7**:

Prove by induction that for all n > 4, Fn > (4/3)n

Then use this result to explain the approximate asymptotic running time of the recursive algorithm for computing the Fibonacci numbers. Is the recursive Fibonacci algorithm fast or slow? Why?

**Answer**

For all n > 1, we have

With n = 2, we have

Or

Which is correct.

Assume the below correct

, we will check with n+1

Indeed we have:

With the recursive algorithm, we can directly execute the recurrence as given in the mathematical definition of the Fibonacci sequence.

However, tt uses Θ(n) stack space and Θ(φn) arithmetic operations, where φ= 1.618 (the golden ratio <https://en.wikipedia.org/wiki/Golden_ratio>).

Time complexity is T(n) = T(n-1) + T(n-2) + C

T(n) = O(2^(n-1)) + O(2^(n-2)) + O(1)

That means, the number of operations to compute F(n) is proportional to the final numerical answer, which grows exponentially.

Reference: <https://en.wikipedia.org/wiki/Fibonacci_numbers>

**Problem 8**:

Algorithm

Input: a non-negative integer n

Output: An array of n unique random numbers, within 0 .. n range.

An empty array *result* (for storing the generated values)

While array filled size < n then

Generate random value *temp* (to call built-in random process)

If temp not in the array then

Add *temp* in the next available slot in the array

Return array *result*.

There are many possible ways to do, in the Java code, I use different data structures for separate approaches.

* With the ArrayList, I have to check if the storing array contains same value as the generated value.
* With HashSet, I don’t need to check, since the HashSet does not allow duplicate value.

Java source code:

Using ArrayList data structure

// Question 8

// Using ArrayList, so we need to check

// before insert the generated number in to the list

// n: the number of generated array

// k: the max range of generated numbers

public static void generateRandomWithArrayList(int n, int k) {

Random ran = new Random();

ArrayList<Integer> arrList = new ArrayList<Integer>();

while (arrList.size() < n) {

int a = ran.nextInt(k + 1);

if (!arrList.contains(a)) {

arrList.add(a);

}

}

System.out.println(arrList);

}

Using HashSet data structure

// Using HashSet because HashSet does not allow duplicate item.

public static void generateRandomWithHashSet(int n, int k) {

Random ran = new Random();

Set<Integer> gen = new HashSet<Integer>();

while (gen.size() < n) {

gen.add(ran.nextInt(k + 1));

}

System.out.println(gen);

}

**Problem 9**:

1. is

Let assume the above is correct, so there would be and,

which meets: for all . Meaning:

Clearly c cannot be constant, which is opposite for the assumption. Hence the statement is false.

1. is

By definition, f is O(g) if there exists a large N and a constant c such that

f(x) < c\*g(x) for all x >= N

In this case it’s true that we have for all big x. However, it’s also true that , and we can choose some c larger than log(3) to end up with

Hence, is .

1. is

By looking at the Wikipedia: <https://en.wikipedia.org/wiki/Logarithm>, **Number Theory** section, we can simply see:

log(n!) = log(n \* (n-1) \* (n-2) \* ... \* 2 \* 1)

= log(n) + log(n-1) + ... + log(2) + log(1)

< log(n) + log(n) + ... + log(n) = n \* log(n)

Now, we check the first half of the above summation, and similarly, we have

log(n!) = log(n) + log(n-1) + ... + log(2) + log(1)

> log(n/2) + ... + log(n) = log(n/2) + log(n/2+1) + ... + log(n-1) + log(n)

>= log(n/2) + ... + log(n/2) = n/2 \* log (n/2)

We also know that log(n!) = Θ(n·log(n)).

Hence with an infinite n number the above statement is correct.

**Problem 10**:

recursiveFactorial

Algorithm recursiveFactorial(n)

Input: A non-negative integer n

Output: n!

if (n = 0 || n = 1) then

return 1

return n \* recursiveFactorial(n-1)

1. Guesing Method to determine the worst-case.

Check how many times the recursiveFactorial() called.

T(1) = 1

T(2) = T(1) + 1 = 2

T(3) = T(2) + 1 = 3

…

T(n) = T(n-1) + 1 = n

In every case, the recursiveFactorial has only n steps, no such worst-case could occur.

1. Prove the correctness of the algorithm

When n = 0 or 1, 1 step, and result = 1

Assume T(k) = k, and k < n

Now consider T(n)

T(n) = T(n-1) + 1 = n - 1 + 1 = n

Therefore, T(n) = n for all n ≥ 1

Each step, the return value is accumulatively multiplied with the previous and backtrack to the base value, 1, when n reduced to 1.

**Problem 11**:

The algorithm:

Algorithm fibonaci(n)

Input: a non-negative integer n

Output: fibonacci value of n

tmp1 🡨 0

tmp2 🡨 0

result 🡨 1

if n is 0

return 0

else

for each i in 1..n do

tmp2 🡨 tmp1 + result

tmp1 🡨 result

result 🡨 tmp2

return result

As we know the Fibonacci value is defined by a summation of the 2 previous values, or

F(2) = F(1) = 1

F(n) = F(n-1) + F(n-2)

Therefore, we will assign the result by 1 from the beginning, that is F(1).

By using the temporary variable tmp1 and tmp2, the values are added and swapped to ensure the result is a summation of the previous 2 values.

For example n = 5, the sequences are:

i = 1, F(1) = result = 1, tmp1 = tmp2 = 0

tmp2 = 0 + 1

tmp1 = result = 1

result = tmp2 = 1

i = 2, result = 1, tmp1 = 1, tmp2 = 1

tmp2 = 1 + 1 = 2

tmp1 = 1

result = tmp2 = 2

i = 3, result = 2, tmp1 = 1, tmp2 = 2

tmp2 = 1+ 2 = 3

tmp1 = 2

result = tmp2 = 3

i = 4, result = 3, tmp1 = 2, tmp2 = 3

tmp2 = 3 + 2 = 5

tmp1 = 3

result = tmp2 = 5

i = 5, result = 5, tmp1 = 3, tmp2 = 5

tmp2 = 3 + 5 = 8

tmp1 = 5

result = tmp2 = 8

The algorithm is correct.

Java source code:

// Question 11

// Iteractive algorithm for Fibonacci

public static void fibonacci(int n) {

int tmp1 = 0;

int tmp2 = 0;

int result = 1;

if (n == 0)

result = 0;

else {

for (int i = 1; i <= n; i++) {

tmp2 = tmp1 + result;

tmp1 = result;

result = tmp2;

}

}

System.out.format("Input number %d, Fibonacci value: %d", n, result);

}

**Problem 12**:

Using substitution method, we can see:

T(n) = \*\*

T(n) = T(n/2) + n

T(n/2) = T(n/4) + n/2

T(n/4) = T(n/8) + n/4

…

T(2) = T(1) + 2 = O (n)

With Master Theorem, from the definition: T(n)=aT(n/b)+f(n)

If f(n) = Ɵ () then

T(n) = Ɵ (n) when a < b

T(n) = Ɵ (n log n) when a = b

T(n) = Ɵ () when a > b

In our case, a = 1, b = 2, f(n) = n, d = 1, we have T(n) = Ɵ (n)

**Problem 13**:

Algorithm

The idea is to take advantage of the fact that the input is sorted (i.e. all 0’s followed by all 1’s). We split the array to two partitions and recur for both.

* If the first element is 1, then all elements are 1, and we return number of elements in that split.
* If the last element of the split is 0, then all elements are 0, and we return 0 for that split.

Input: an array, left index, right index

Output: number of 1

If element of right index = 0 then

return 0

If element of left index = 1 then

return array length = right index - left index + 1

middle index for splitting purpose 🡨 (left index + right index) / 2

return the sum of the recurring algorithm on left partition of the checking array between left index and middle index and right partition the checking array array between middle index and right index

Java source code for the algorithm implementation

// Question 13

// Assume that the given array is sorted in ascending order from left to right

// If not sorted, use sortBinaryArray

// rIdx = Right Index

// lIdx = Left Index

public static int countOnes(int[] arr, int lIdx, int rIdx) {

// If last element of the array is 0, no 1 in the array

if (arr[rIdx] == 0)

return 0;

// If first element of the array is 1, return the length of the checking array

// which is (rightIndex - leftIndex + 1)

if (arr[lIdx] == 1)

return rIdx - lIdx + 1;

// Apply divide and conquer approach and recur

int middle = (lIdx + rIdx) / 2;

return countOnes(arr, lIdx, middle) + countOnes(arr, middle + 1, rIdx);

}

**Explanation**:

As the given array is sorted, so we only need to find the position of the very first one from left to right direction. There are two approaches:

1. If we start from the very left and or right, we have to go to almost all items (all items if there’s non-one item). This is O(n) time.
2. If we apply the divide-conquer method here, which means we will split the array into two parts, then recursively repeat same way to each split, we will very soon locate the position of the one (1). With this approach, actually our time complexity is n/2, that less than O(n) as the above.